Supplemental Methods

Theory

Time-complexity

We introduced the definition of the entropy-scaling similarity search data structure in Figure 1. For ease of analysis, we will work in a high-dimensional metric space and consider the database as a set D of n unique points in that metric space. Define $B_S(q, r) = \{p \in S : ||q - p|| < r\}$. The similarity search problem is thus to compute $B_D(q, r)$ for a query q and radius r. Note however that the metricity requirement is needed only for a 100% sensitivity guarantee; other distance functions can be used, but result in some loss in sensitivity. However, regardless of the distance function chosen, there cannot be a loss of specificity; false positives will never be introduced because the fine search is just the original search function on a smaller subset of the database.

A set C of k cluster centers are chosen such that no cluster has radius greater than a user-specified parameter r_c and no two cluster centers are within distance r_c of one another. The data structure then clusters the points in the set by assigning them to their nearest cluster center. Overloading notation a bit, we will identify each cluster with its center, so C is also the set of clusters. For a given similarity search query for all items within distance r of a query q, this data structure breaks the query into coarse and fine search stages. The coarse search is over the list of cluster centers,

returning $B_C(q, r + r_c)$. Let

$$F = \bigcup_{c \in B_C(q, r+r_c)} c,$$

the union of all the returned clusters. By the Triangle Inequality, $B_D(q, r) \subseteq F$, which combined with $F \subseteq D$ implies that $B_F(q, r) = B_D(q, r)$. Thus, a fine search over the set F will return all items within radius r of q.

Note that we require the metricity requirement only for the Triangle Inequality. It turns out that many interesting distance functions are not metrics, but still almost satisfy the Triangle Inequality, which is nearly sufficient. More precisely, if a fraction α of the triples in S do not satisfy the Triangle Inequality, then in expectation, we will have sensitivity $1 - \alpha$. As shown in the results, empirically, this loss in sensitivity appears to be low and can likely be ameliorated by increasing the coarse search radius.

Provided the fractal dimension of the database is low, this data structure allows for similarity search queries in time roughly linear in the metric entropy of the database. Additionally, without increasing the asymptotic time-complexity, this data structure can also be stored in an *information theoretic* entropy-compressed form.

Note that entropy-scaling data structures are distinct from both succinct data structures and compressed data structures. Succinct data structures are ones that use space close to the information-theoretic limit in the worst case while permitting efficient queries; i.e. succinct data structures do not depend on the actual entropy of the underlying data set, but have size-dependence on the potential worst-case entropy of the data set (Jacobson, 1988). Compressed (and opportunistic) data structures, on the other hand,

bound the amount of the space used by the entropy of the data set while permitting efficient queries (Grossi & Vitter, 2005; Ferragina & Manzini, 2000). Entropy-scaling data structures are compressed data structures, but are distinct, as unlike entropy-scaling data structures, compressed data structures do not measure time-complexity in terms of metric entropy. Additionally, existing compressed data structures such as the compressed suffix array and the FM-index are designed for the problem of pattern matching (Grossi & Vitter, 2005; Ferragina & Manzini, 2000). While related to similarity search, pattern matching does not admit as general of a notion of distance as the similarity search problem. While compressed sensing has also been applied to the problem of finding a representative set of genes for a collection of expression samples (Prat et al., 2011), compressed sensing is distinct from entropy-scaling data structures.

The primary advance of entropy-scaling data structures is that they bound both space and time as functions of the data set entropy (albeit using two different notions of entropy).

Complexity bounds

We first define the concept of metric entropy and entropy dimension in the usual manner:

Definition 1 ((Tao, 2008) Definition 6.1)**.** Let X be a metric space, let D be a subset of X, and let $\rho > 0$ be a radius.

- The *metric entropy* $N_\rho(D)$ is the fewest number of points $x_1, \ldots, x_n \in D$ such that the balls $B(x_1, \rho), \ldots B(x_n, \rho)$ cover D.

Definition 2 ((Falconer, 1990))**.** The Hausdorff dimension of a set D is given by

$$\dim_{\text{Hausdorff}}(D) := \lim_{\rho \to 0} \frac{\log N_\rho(D)}{\log 1/\rho}$$

Unfortunately, as D is a finite, discrete, set, the given definision always gives $\dim_{\text{Hausdorff}}(D) = 0$. However, we are only interested in scaling behaviors around large radii, so instead we use:

Definition 3. Define fractal dimension d of a set D at a scale $[\rho_1, \rho_2]$ by

$$d = \max_{\rho \in [\rho_1, \rho_2]} \left\{ \frac{\log \frac{N_\rho(D)}{N_{\rho_1}(D)}}{\log \frac{\rho}{\rho_1}} \right\}$$

Intuitively, this means that when we double the radii, the metric entropy, or number of covering spheres needed, decreases by a multiplicative factor of 2^d. On average, this also means that when we double the radius of a sphere around a point, the number of points in the larger sphere is roughly the number of points in the smaller sphere multiplied by 2^d, because otherwise the spheres could not cover the space. This latter behavior is what we measure when we talk about local fractal dimension around a point in the main paper.

Recall that k entries are selected as cluster centers for partitioning the database to result in clusters with maximum radius r_c. From the definition above, when setting $\rho = r_c$, it is trivial to verify $k \leq N_{r_c}(D)$. This upper bound is guaranteed by our requirement that the cluster centers not be within distance r_c.

Given any query q, the coarse search over the cluster centers always requires k comparisons. Additionally, the fine search is over the set F, defined to be the union of clusters with centers within distance $r + r_c$ from q. As the

time-complexity of similarity search is just the total of the coarse and fine searches, this implies that the total search time is $O(k + |F|)$.

By the triangle inequality, $F \subset B_D(q, r + 2r_c)$, so we can bound $|F| \leq |B_D(q, r + 2r_c)|$. Let the fractal dimension D at the scale between r_c and $2r_c + r$ be d. Recall that the local fractal dimension determines how many more points we hit when we scale the radius of a sphere. Then in expectation over possible queries q,

$$|B_D(q, r + 2r_c)| \sim |B_D(q, r)| \left(\frac{r + 2r_c}{r} \right)^d,$$

because we are measuring the relative number of points found in spheres of radius r vs radius $r + 2r_c$ respectively. Thus, total search time is

$$O\left(k + |B_D(q, r)| \left(\frac{r + 2r_c}{r} \right)^d \right).$$

However, note that k is linear in metric entropy and $|B_D(q, r)|$ is the output size, so similarity search can be performed in time linear to metric entropy and a polynomial factor of output size. Provided that the fractal dimension d is small and k is large, the search time will be dominated by the metric entropy component, which turns out to be the regime of greatest interest for us. We have thus proven bounds for the time-complexity of similarity search.

Space-complexity

Here we relate the space-complexity of our entropy-scaling similarity search data structure to information-theoretic entropy. Traditionally, information-theoretic entropy is a measure of the uncertainty of a distribution or random variable and is not well-defined for a finite database. However, the notion of

information-theoretic entropy is often used in data compression as a shorthand for the number of bits needed to encode the database, or a measure of the randomness of that database. We use entropy in the former sense; precisely, we define the entropy of a database as the number of bits needed to encode that database, a standard practice in the field. Thus, we consider entropy-compressed forms of the original database, such as that obtained by Prediction by Partial Matching (PPM), Lempel-Ziv compression (e.g. Gzip), or a Burrows-Wheeler Transform (as in Bzip2), and use their size as an estimate of the entropy S_{orig} of the database.

For all commonly used compression techniques, decompression time is linear in the size of the uncompressed data. Obviously, even with linear decompression, decompressing the entire database for each similarity search would squander the entropy-scaling benefits of our approach. However, note that the fine search detailed above only needs access to a subset of clusters and furthermore needs full access to that set of clusters. It is therefore always asymptotically 'free' to decompress an entire cluster at once, if any member of that cluster needs to accessed. Thus, one ready solution is to simply store entropy-compressed forms of each cluster separately.

Compressing each cluster separately preserves runtime bounds, but makes it difficult to compare the compressed clustered database size to the original compressed database size. This results from the possibility that redundancy across clusters that would originally have been exploited by the compressor can no longer be exploited once the database is partitioned. Intuitively, for any fixed-window or block compressor, grouping together similar items into clusters should increase the performance of the compressor, but it is unclear

a priori if that balances out the loss of redundancy across clusters.

A somewhat more sophisticated solution is to reorder the entries of the database by cluster, compress the entire database, and then store indexes into the starting offset of each cluster. For popular tools such as Gzip or Bzip2, this is possible with constant overhead κ per index. Because the entire database is still being compressed, redundancy across clusters can be exploited to reduce compressed size, while still taking advantage of similar items being grouped together. Thus, in expectation over uniformly-randomly chosen orderings of the database entries (obviously, there is some optimal ordering, but computing that is computationally infeasible), the compressed clustered database size $S_{clust} \leq S_{orig}$. Then, total expected space-complexity of our data structure is $O(\kappa k + S_{orig})$; recall here that k is the number of clusters and is bounded by the metric entropy of the database. Thus, space complexity is linear in metric entropy plus information-theoretic entropy.

Additionally, given that our distance function measures marginal information-theoretic entropies, we can also give a bound on the total information-theoretic entropy of the database by using metric entropy and the cluster radius. Let l be the maximum distance of two points in the space. The naïve upper bound on total entropy is then $O(nl)$, where n is the total number of points in the database, because distance and entropy are related. Recall that we chose k points as cluster centers, where k is bounded by metric entropy, for a maximum cluster radius r_c. Encoding each non-center point p as a function of the nearest cluster center requires $O(nr_c)$ bits. Specifying the privileged points again requires $O(kl)$ bits, so together the total information-theoretic entropy is $O(kl + nr_c)$. In other words, not only is space complexity linear in

metric entropy plus information-theoretic entropy, but information-theoretic entropy itself is also bounded by the low-dimensional coarse structure of the database.

Clustering time complexity

Although clustering the database is a one-time cost that can further be amortized over future queries, we still require that cluster generation be tractable. Here we present a trivial $O(kn)$ algorithm for cluster generation with clusters of maximum radius r_c that appears to work sufficiently well in practice (and is the algorithm used in esFragBag):

- Initialize an empty set of cluster centers C. Let $\delta(x, C)$ be the distance from a point x to C, defined to be ∞ if $C = \emptyset$.
- Randomly order the n entries of the database $D = \{d_1, \ldots, d_n\}$
- For $i = 1, \ldots n$,
  - If $\delta(d_i, C) > r_c$, append d_i to C.
- For $i = 1, \ldots n$,
  - Assign d_i to the cluster represented by the nearest item in C.

Because we need to compare each of n items against up to k items in C in each of the for loops, this trivial algorithm takes $O(kn)$ time.

Additionally, insertions can be performed in $O(k)$ time. For a new entry d_{n+1}, if $\delta(d_{n+1}, C) \leq r_c$, assign d_{n+1} to the cluster represented by the nearest item in C. Otherwise, append d_{n+1} to C as a new cluster center. This clearly requires exactly k comparisons to do. Note that with insertions of this kind,

items are no longer guaranteed to be assigned to the nearest cluster center; however, they are still guaranteed to be assigned to some cluster center within distance r_c, which is all that is needed for entropy-scaling to work.

Deletions are slightly more complicated. If the entry to be deleted is not a cluster center, then removing it takes constant time. However, if it is a cluster center, we effectively have to remove the entire cluster and reinsert all the non-center elements, which will take $O(k \cdot [\text{size of cluster}])$. Thus, in expectation over a uniform random choice of item to be deleted, deletions can also be performed in $O(k)$ time.

Ammolite

Simplification and compression

Given a molecular graph, any vertex or edge that is not part of a simple cycle or a tree is removed, and any edge that is part of a tree is removed. This preserves the node count, but not the topology, of tree-like structures, and preserves simple cycles, which represent rings in chemical compounds. For example, as shown in Figure S2, both caffeine and adenine would be reduced to a purine-like graph.

After this transformation is applied to each molecule in a database to be compressed, we identify all clusters of fully-isomorphic transformed molecular graphs. Isomorphism detection is performed using the VF2 (Cordella et al., 2001) algorithm; a simple hash computed from the number of vertices and edges in each transformed molecular graph is first used to filter molecular graphs that cannot possibly be isomorphic. A representative from each such cluster is stored in SDF format; collectively, these representatives form

a "coarse" database. Along with each representative, we preserve the information necessary to reconstruct each original molecule, as a pointer to a set of vertices and edges that have been removed or unlabeled.

Ammolite is implemented in Java, and its source code is available on Github.

MICA

Alphabet Reduction

Alphabet reduction—reducing the 20-letter standard amino acid alphabet to a smaller set, in order to accelerate search or improve homology detection—has been proposed and implemented several times (Bacardit et al., 2007; Peterson et al., 2009). In particular, Murphy et al. (2000) considered reducing the amino-acid alphabet to 17, 10, or even 4 letters. More recently, Zhao et al. (2012) and Huson & Xie (2013) applied a reduction to a 4-letter alphabet, termed a "pseudoDNA" alphabet, in sequence alignment.

When using BLASTX for coarse search, we extend the compression approach of Daniels et al. (2013) using a reversible alphabet reduction. We use the alphabet reduction of Murphy et al. (2000) to map the standard amino acid alphabet (along with the four common ambiguous letters) onto a 4-letter alphabet. Specifically, we map F, W, and Y into one cluster; C, I, L, M, V, and J into a second cluster, A, G, P, S, and T into a third cluster, and D, E, N, Q, K, R, H, B, and Z into a fourth cluster. By storing the offset of the original letter within each cluster, the original sequence can be reconstructed, making this a reversible reduction. This alphabet reduction is not used when using DIAMOND for coarse search, as DIAMOND already

relies on its own alphabet reduction.

Database Compression

Given a protein sequence database to be compressed, we proceed as follows:

1. First, initialize a table of all possible k-mer seeds of our (possibly 4-letter reduced) alphabet, as well as a coarse database of sequences, initially containing the (possibly reduced-alphabet) first sequence in the input database.
2. For each k-mer of the first sequence, store its position in the corresponding entry in the seed table.
3. For each subsequent sequence s in the input, reduce its alphabet and slide a window of length k along the sequence, skipping single-letter repeats of length greater than 10.
4. (a) Look up these k residues in the seed table. For every entry matching to that k-mer in the seed table, follow it to a corresponding subsequence in the coarse database and attempt *extension* (defined below). If no subsequences from this window can be extended, move the window by m positions, where m defaults to 20.
   (b) If a match was found via extension, move the k-mer window to the first k-mer in s after the match, and repeat the extension process.

Given a k-mer in common between sequence s and a subsequence s' pointed to by the seed table, first attempt *ungapped* extension:

1. Within each window of length m beginning with a k-mer match, if there are at least 60% matches between s and s', then there is an ungapped match.
2. Continue ungapped matching using m-mer windows until no more m-mers of at least 60% sequence identity are found.
3. The result of ungapped extension is that there is an alignment between s and s' where the only differences are substitutions, at least 60% of the positions contain exact matches.

When ungapped extension terminates, attempt *gapped* extension. From the end of the aligned regions thus far, align 25-mer windows of both s and s' using the Needleman-Wunsch (Needleman & Wunsch, 1970) algorithm using an identity matrix. Note that the original caBLASTP (Daniels et al., 2013) used BLOSUM62 as it was operating in amino acid space; as we are now operating in a reduced-alphabet space, an identity matrix is appropriate, just as it is for nucleotide space. After gapped extension on a window length of 25, attempt ungapped extension again.

If neither gapped nor ungapped extension can continue, end the extension phase. If the resulting alignment has less than 70% sequence identity (in the reduced-alphabet space), or is shorter than 40 residues, discard it, and attempt extension on the next entry in the seed table for the original k-mer, continuing on to the next k-mer if there are no more entries.

If the resulting alignment does have at least 70% sequence identity in the reduced-alphabet space, and is at least 40 residues long, then create a link from the entry for s' in the coarse database to the subsequence of s corresponding to the alignment. If there are unaligned ends of s shorter than

30 residues, append them to the match. Longer unaligned ends that did not match any subsequences reachable from the seed table are added into the coarse database themselves, following the same k-mer indexing procedure as the first sequence.

Finally, in order to be able to recover the original sequence with its original amino acid identities, a *difference script* is associated with each link. This difference script is a representation of the insertions, deletions, and substitutions resulting from the Needleman-Wunsch alignment, along with (if alphabet reduction is used) the offset in each reduced-alphabet cluster needed to recover the original alphabet. Thus, for example, a valine (V) is in the cluster containing C, I, L, M, V, and J. Since it is the 4th entry in that 5-entry cluster, we can represent it with the offset 4. Since the largest cluster contains 9 elements, only four bits are needed to store one entry in the difference script. More balanced clusters would have allowed 3-bit storage, but at the expense of clusters that less faithfully represented the BLOSUM62 matrix and the physicochemical properties of the constituent amino acids.

Because of the seed table, compression is memory-intensive and CPU-intensive. Compressing the September, 2014 NCBI NR database required approximately 39 hours on a 12-core Xeon with 128GB RAM.

Query Clustering

Metagenomic reads are themselves nucleotide sequences, so no alphabet reduction is performed on them directly. When BLASTX is used for coarse search, MICA relies on query-side clustering. Metagenomic reads are compressed using the same approach as the protein database, without the alphabet reduction step and with a number of different parameters. The difference

scripts for metagenomic reads do not rely on the cluster offsets, but simply store the substituted nucleotides.

Furthermore, unlike protein databases, where most typical sequences range in length from 100 to over 1000 amino acids, next-generation sequencing reads are typically short and usually of fixed length, which is known in advance. Thus, the minimum alignment length required for a match, and the maximum length unaligned fragment to append to a match, require different values based on the read length.

An additional complication is that insertions and deletions from one read to another will change the reading frame, potentially resulting in different amino acid sequences. For this reason, query clustering requires long, *ungapped* windows of high sequence identity. Specifically, for 202-nucleotide reads, for two sequences to cluster together, we require a 150-nucleotide ungapped region of at least 80% sequence identity.

We note that unlike the compression of the database, which can be amortized over future queries, the time spent clustering and compressing the queries cannot be amortized. Thus, we would not refer to the query clustering as entropy-scaling, but it still provides a constant speed-up. For this reason, we include the time spent clustering and compressing queries in the search time for MICA.

Search

Given a compressed protein database and a compressed query read set, search comprises two phases. The first, *coarse search*, considers only the coarse sequences—the representatives—resulting from compression of the protein database and the query set. When BLASTX is used for the coarse

search, each coarse nucleotide read is transformed into each of the six possible amino acid sequences that could result from it (three reading frames for both the sequence and its reverse complement). Then, each of these amino acid sequences is then reduced back to a four-letter alphabet using the same mapping as for protein database compression. For convenience, the four-letter alphabet is represented using the standard nucleotide bases, though this has no particular biological significance. This is done so that the coarse search can rely on BLASTN (nucleotide BLAST) to search these sequences against the compressed protein database.

For each coarse query representative (identified using a a coarse E-value of 1000, along with the BLASTN arguments `-task blastn-short -penalty -1`; these arguments are recommended by the NCBI BLAST+ manual when queries are short), the set of coarse hits is used to reconstruct all corresponding sequences from the original database by following links to original sequence matches and applying their difference scripts. The resulting *candidates* are thus original sequences from the protein database, in their original amino acid alphabet. The query representative is also used to reconstruct all corresponding sequences from the original read set. Thus, for each coarse query representative, there is now a subset of the metagenomic read set (the reads represented by that coarse query) and also a subset of the protein database (the candidates).

When DIAMOND is used for coarse search, instead of an E-value threshold, the argument `--top 60` is used; this causes DIAMOND to return all coarse hits whose score is within 60% of the top-scoring hit. Without this argument, DIAMOND defaults to returning at most 25 hits for each query

sequence, which would result in significant loss of recall.

The second phase, *fine search*, uses standard DIAMOND or BLASTX to translate each of these reads associated with a coarse query representative and search for hits only in the subset of the database comprising the candidates. This fine search phase relies on a user-specified E-value threshold (or other user-specified parameters to DIAMOND or BLASTX) to filter hits. To ensure that E-value calculation is correct, BLASTX uses a corrected database size which is the size of the original, uncompressed protein database.

Benchmarking

Although our primary result is the direct acceleration of BLASTX using our entropy-scaling data structures, we also compared MICA to RapSearch2 (Zhao et al., 2012) version 2.22 and the November 29, 2014 version of DIAMOND (Buchfink et al., 2015). All tests were performed on a 12-core Intel Xeon X5690 running at 3.47GHz with 88GB RAM and hyperthreading; 24 threads were allowed for all programs. Diamond was run with the `--sensitive` option. In all cases, an E-value threshold of `1e-7` was used.

For the raw-read dataset, we filtered out reads starting or ending with 10 or more no-calls ('N').

MICA is implemented in Go, and its source code is available on Github.

esFragBag

We took the existing FragBag method as a black box and by design did not do anything clever in esFragBag except apply the entropy-scaling similarity search data structure. We used a Go language implementation of FragBag, written by Andrew Gallant. Additionally, we removed the sorting-by-

distance feature of Andrew Gallant's FragBag search implementation, which does not improve the all-matching results we were interested in here—it lowers k-nearest neighbor search memory requirements while dominating the running time of ρ-nearest neighbor, the problem at hand. This was done for both the FragBag and the esFragBag benchmarks, to ensure comparability. All code was written in Go, and is available on Github.

The entire 2014 Oct 31 version of the Protein Data Bank was downloaded and the database was composed of fragment frequency vectors generated from all of the relevant PDB files using the 400-11.json fragment list (Budowski-Tal et al., 2010). For this paper, we implemented the benchmarking in Go, and have provided the source code for the benchmarking routine on Github. This allowed us to benchmark just the search time, excluding the time to load the database from disk. Note that the prototype implementation of esFragBag available only supports the all ρ-nearest neighbor search query found in FragBag.

Supplemental References

Bacardit, J., Stout, M., Hirst, J. D., Sastry, K., Llorà, X., & Krasnogor, N. (2007). Automated alphabet reduction method with evolutionary algorithms for protein structure prediction. In Proceedings of the 9th annual conference on Genetic and evolutionary computation (pp. 346–353). ACM.

Buchfink, B., Xie, C., & Huson, D. H. (2015). Fast and sensitive protein alignment using DIAMOND. Nature methods, 12, 59–60.

Budowski-Tal, I., Nov, Y., & Kolodny, R. (2010). FragBag, an accurate

representation of protein structure, retrieves structural neighbors from the entire PDB quickly and accurately. Proceedings of the National Academy of Sciences, 107, 3481–3486.

Cordella, L. P., Foggia, P., Sansone, C., & Vento, M. (2001). An improved algorithm for matching large graphs. In 3rd IAPR-TC15 workshop on graph-based representations in pattern recognition (pp. 149–159).

Daniels, N. M., Gallant, A., Peng, J., Cowen, L. J., Baym, M., & Berger, B. (2013). Compressive genomics for protein databases. Bioinformatics, 29, i283–i290.

Falconer, K. (1990). *Fractal geometry: mathematical foundations and applications*. John Wiley & Sons.

Ferragina, P., & Manzini, G. (2000). Opportunistic data structures with applications. In Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on (pp. 390–398). IEEE.

Grossi, R., & Vitter, J. S. (2005). Compressed suffix arrays and suffix trees with applications to text indexing and string matching. SIAM Journal on Computing, 35, 378–407.

Huson, D. H., & Xie, C. (2013). A poor man's BLASTX-high-throughput metagenomic protein database search using PAUDA. Bioinformatics, (p. btt254).

Jacobson, G. J. (1988). *Succinct static data structures*. Ph.D. thesis Carnegie Mellon University.

Murphy, L. R., Wallqvist, A., & Levy, R. M. (2000). Simplified amino acid alphabets for protein fold recognition and implications for folding. Protein Engineering, 13, 149–152.

Needleman, S. B., & Wunsch, C. D. (1970). A general method applicable to the search for similarities in the amino acid sequence of two proteins. Journal of molecular biology, 48, 443–453.

Peterson, E. L., Kondev, J., Theriot, J. A., & Phillips, R. (2009). Reduced amino acid alphabets exhibit an improved sensitivity and selectivity in fold assignment. Bioinformatics, 25, 1356–1362.

Prat, Y., Fromer, M., Linial, N., & Linial, M. (2011). Recovering key biological constituents through sparse representation of gene expression. Bioinformatics, 27, 655–661.

Tao, T. (2008). Product set estimates for non-commutative groups. Combinatorica, 28, 547–594.

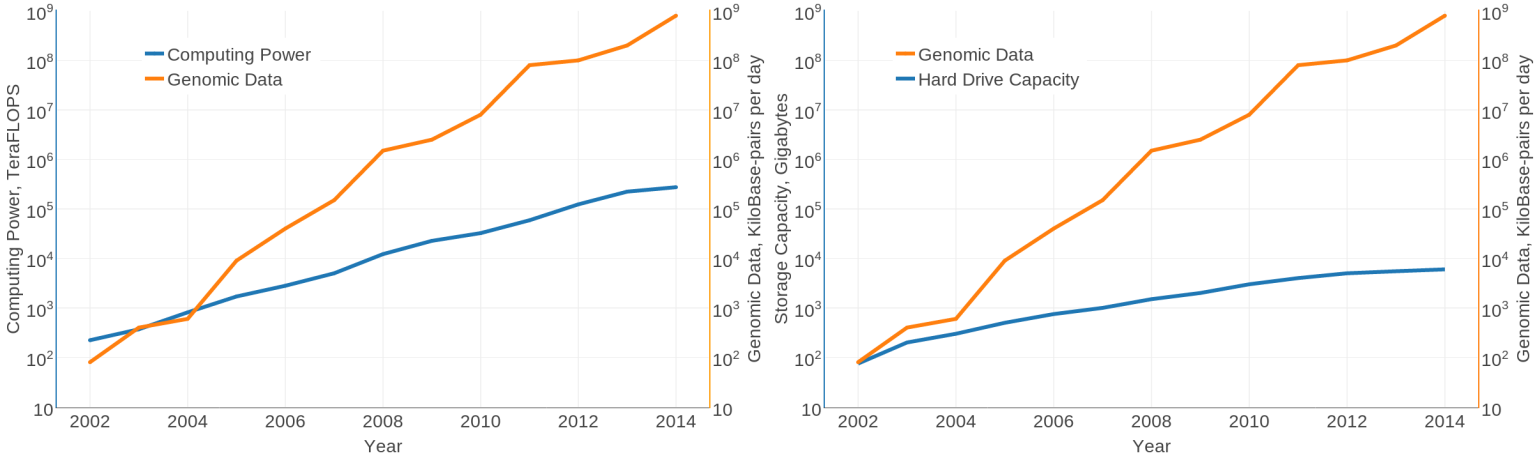
Zhao, Y., Tang, H., & Ye, Y. (2012). RAPSearch2: a fast and memory-efficient protein similarity search tool for next-generation sequencing data. Bioinformatics, 28, 125–126.

Supplemental Figures

S1 Genomic data available has grown at a faster exponential rate than computer processing power and disk storage. These plots represent, on a log scale, the daily growth in sequence data from GenBank along with (a) the combined computing power (in TeraFLOPs) of the Top 500 Supercomputer list, and (b) the largest commercially-available hard disk drives.

S2 **(Related to Table 2)** Ammolite's preprocessing during the clustering phase. Ammolite removes nodes and edges that do not participate in simple cycles, and treats all edges as simple, unlabeled edges. In this example, both caffeine and adenine become a purine-like graph structure. Note that the resulting graph has no implicit hydrogens.

S3 **(Related to Figure 3)**Local fractal dimension at different scales for the space of PDB FragBag frequency vectors. Each data point is defined by dimension $d = \frac{\log(n_2/n_1)}{\log(r_2/r_1)}$, where n_1, n_2 are the number of similarity search hits within radius respectively r_1, r_2, and $r_2 - r_1$ is the increment size of 0.01 for cosine distance and 1 for euclidean distance. In most regimes, local fractal dimension is consistently low, except for a large spike when radius expands to include the central cluster of proteins. esFragBag achieves the most acceleration when both output size is small and we remain in a low fractal dimension regime.



(a) **(b)**

Figure S1: Genomic data available has grown at a faster exponential rate than computer processing power and disk storage. These plots represent, on a log scale, the daily growth in sequence data from GenBank along with (a) the combined computing power (in TeraFLOPs) of the Top 500 Supercomputer list, and (b) the largest commercially-available hard disk drives.

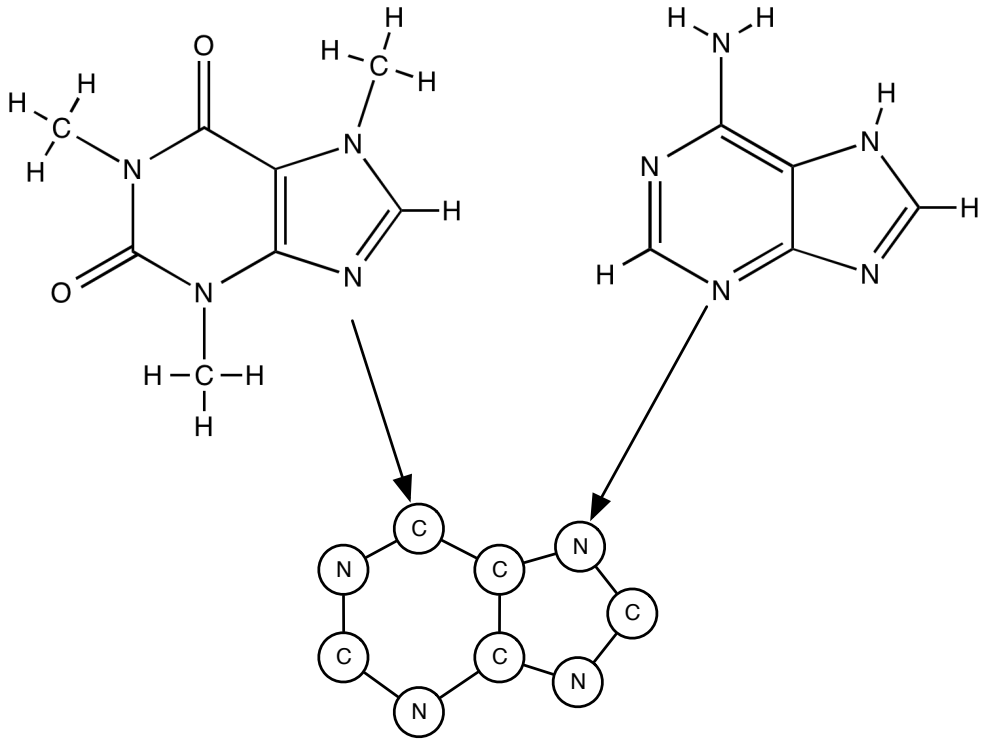


Figure S2: (Related to Table 2) Ammolite's preprocessing during the clustering phase. Ammolite removes nodes and edges that do not participate in simple cycles, and treats all edges as simple, unlabeled edges. In this example, both caffeine and adenine become a purine-like graph structure. Note that the resulting graph has no implicit hydrogens.

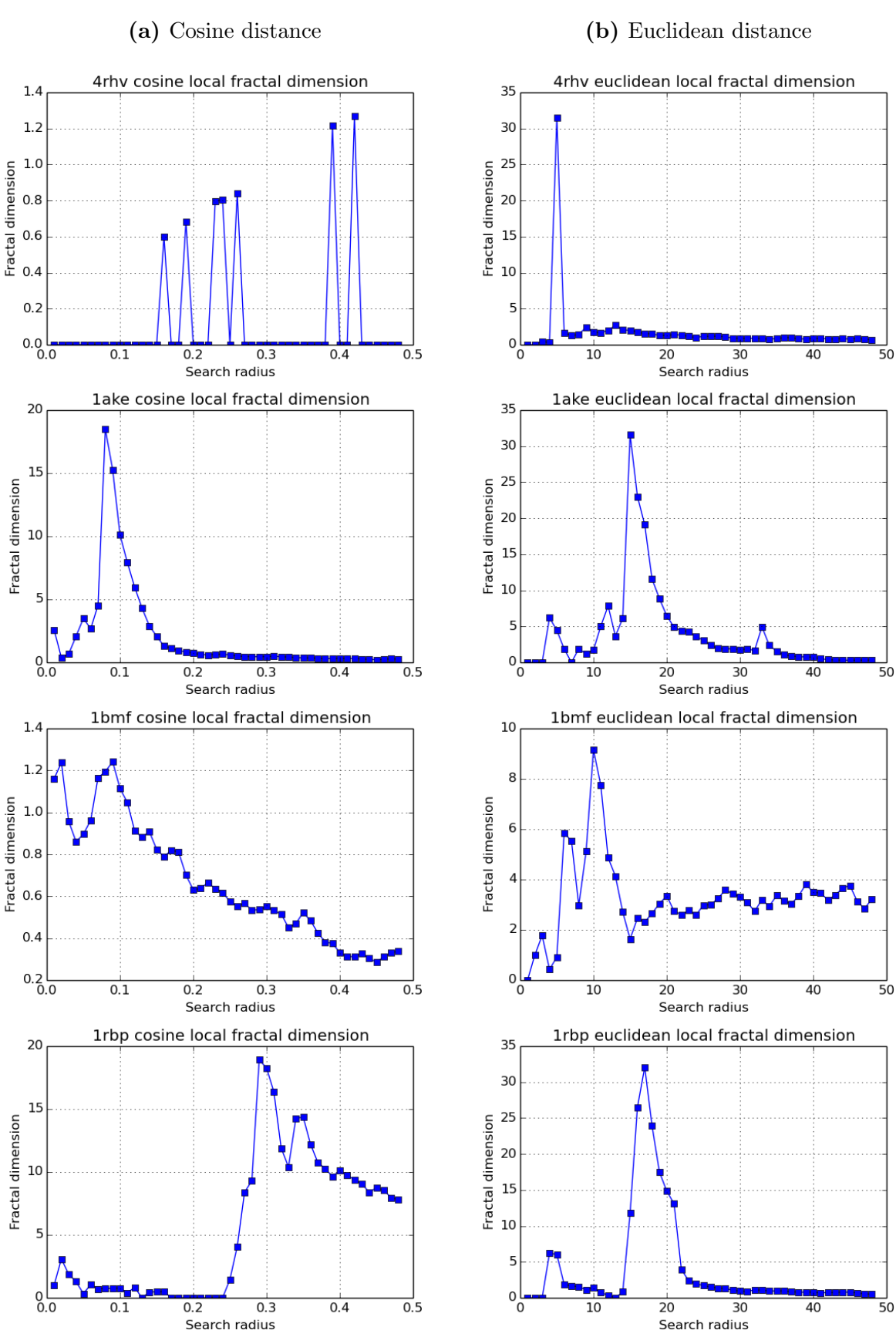


Figure S3: (Related to Figure 3) Local fractal dimension at different scales for the space of PDB FragBag frequency vectors. Each data point is defined by dimension $d = \frac{\log(n_2/n_1)}{\log(r_2/r_1)}$, where n_1, n_2 are the number of similarity search hits within radius respectively r_1, r_2, and $r_2 - r_1$ is the increment size of 0.01 for cosine distance and 1 for euclidean distance. In most regimes, local fractal dimension is consistently low, except for a large spike when radius expands to include the central cluster of proteins. esFragBag achieves the most acceleration when both output size is small and we remain in a low fractal dimension regime.